NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AUTOMATIC INFERENCE OF CRYPTOGRAPHIC KEY LENGTH BASED ON ANALYSIS OF PROOF TIGHTNESS

by

Derek L. Swenningsen

June 2016

Thesis Advisor: Mark Gondree
Second Reader: George Dinolt

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE June 2016	3. REPORT TYPE AND DATES COVERED Master's Thesis 07-01-2015 to 06-17-2016	

4. TITLE AND SUBTITLE

AUTOMATIC INFERENCE OF CRYPTOGRAPHIC KEY LENGTH BASED ON ANALYSIS OF PROOF TIGHTNESS

5. FUNDING NUMBERS

6. AUTHOR(S)

Derek L. Swenningsen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Currently, reasoning about key lengths within a security scheme involves utilizing generalized recommendations or conducting lengthy manual analyses of how security parameters relate to the security of the scheme. In this paper, we provide the tools necessary for automating reasoning about key lengths and effective security within a security scheme. We first formalize the reasoning about cryptographic proofs within an attack tree structure, then expand attack tree methodology to include cryptographic reductions. We then provide the algorithms for maintaining and automatically reasoning about these expanded attack trees. We provide a software tool that utilizes machine-readable proof and attack metadata and the attack tree methodology to provide rapid and precise answers regarding security parameters and effective security. This eliminates the need to rely on generalized recommendations and provides timely reanalysis when newfound attacks or proofs surface. We validate our software tool within the Schnorr public-key signature scheme as a case study.

14. SUBJECT TERMS

keylength analysis, automated reasoning, attack tree, attack tree analysis, cryptographic reasoning

15. NUMBER OF PAGES   65

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

i

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AUTOMATIC INFERENCE OF CRYPTOGRAPHIC KEY LENGTH BASED ON ANALYSIS OF PROOF TIGHTNESS

Derek L. Swenningsen
Major, United States Marine Corps
B.S., Embry-Riddle Aeronautical University, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2016

Approved by: Mark Gondree
Thesis Advisor

George Dinolt
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Currently, reasoning about key lengths within a security scheme involves utilizing generalized recommendations or conducting lengthy manual analyses of how security parameters relate to the security of the scheme. In this paper, we provide the tools necessary for automating reasoning about key lengths and effective security within a security scheme. We first formalize the reasoning about cryptographic proofs within an attack tree structure, then expand attack tree methodology to include cryptographic reductions. We then provide the algorithms for maintaining and automatically reasoning about these expanded attack trees. We provide a software tool that utilizes machine-readable proof and attack metadata and the attack tree methodology to provide rapid and precise answers regarding security parameters and effective security. This eliminates the need to rely on generalized recommendations and provides timely reanalysis when newfound attacks or proofs surface. We validate our software tool within the Schnorr public-key signature scheme as a case study.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Acronyms and Abbreviations

BSI	Bundesamt fur Sicherheit in der Informationstechnik
EF-ACMA	existential forgery under an adaptively chosen-message attack
GMC	guaranteed minimum cost
GNFS	general number field sieve
GPC	guaranteed peak cost
NIST	National Institute of Standards and Technology
UML	Unified Modeling Language

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

There are many people without whom I would not have been able to produce this thesis. First, I extend the richest gratitude to my advisor, Dr. Mark Gondree, with whom it was a distinct honor to work. Romans 12:6-7 says, "We have different gifts, according to the grace given to each of us. If your gift is prophesying, then prophesy in accordance with your faith; if it is serving, then serve; if it is teaching, then teach." I firmly believe Dr. Gondree's grace-given gift is teaching. Throughout this endeavor, I found myself thanking God for Mark's patience when I was troubled, understanding when I was confused, and dedication to ensuring I grew in my knowledge and progressed toward attaining my goals. I know he guided me under the principles of Psalm 32:8, which says, "I will instruct you and teach you in the way you should go; I will counsel you with my loving eye on you." Dr. Gondree is a phenomenal counselor and I wish him the best as he sets out to further his career. Mark, may you find your new home an enriching environment in which your career will flourish.

To my second reader, Dr. George Dinolt, I extend my warmest thanks. Dr. Dinolt's joy in his work is infectious, his input pivotal, and his commitment perpetual. Throughout my research, I found myself going to his office for professional advice and staying too long exchanging life stories about family, friends, and our shared joy, flying. Sir, I thank you for your advice in both academia and life.

To my son, Braden, I give my deepest thanks. Braden always understood and never complained when I needed to put in a few extra hours of work. At only 11, but with the maturity of a grown man, he grounded me by reminding me to stop, enjoy life, and play a little. After a few hours with him, I was rejuvenated and ready to forge on. Braden, I could not be more proud of you than I am. You are already on your way to becoming a better man than I could ever be.

Finally, to my wife Lisa, my rock, I love you. Over the past 13 years, through all the workups and Waffle Houses, deployments and natural disasters, late-night flights and last-minute duties, you have led our family with love, grace, strength, and dignity. I thank you for being there not only for me, but for our son, too. From but a glance of your smile, I am lifted and inspired. Without you, and the inspiration you sparked in me, none of this was possible.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Currently, when an organization reasons about keylength, it is left to choose from an array of sometimes conflicting recommendations provided by several organizations. These recommendations are typically generalized and furnish conservative values. Furthermore, the organizations that maintain them tend to update them with a periodicity of a year or more regardless of newly published papers or newly found attacks that may affect them.

The alternative to this method involves a cryptographer reasoning about keylength through an analysis relating security parameters of a particular scheme to its effective security. This is done with security proofs (typically manual) that relate the parameters and effective security. This method is less general than the recommendations and can be updated as often as one can afford; however, it can be arduous and requires experts to complete.

We explore developing a software tool that can automate the reasoning about keylength for a cryptographic scheme. This software tool applies machine-readable proof and attack metadata using an attack tree approach to provide timely and accurate answers concerning security parameters and effective security. Being derived from an attack tree methodology, this approach naturally inherits extensibility and modularity. This research expands traditional attack tree analysis to consider symbolic constraints related to proof tightness and attack cost. This allows experts in one domain (e.g., attacks employing the general number field sieve) to inform experts from another domain (e.g., the relationship between the subgroup discrete log problem and some, specific hybrid signature scheme) by supplying new modules with appropriate metadata. This makes this type of reasoning significantly more dynamic. Its modular design allows organizations to incorporate new attack metadata and make prompt, informed reactions to new attacks or security announcements. Furthermore, precisely and immediately re-generating analyses related to security parameters enables counter-factual reasoning about these proofs: would improving the tightness bounds of prior work effectively strengthen the security of a scheme (or does some other factor create a bottleneck)? Given some selection parameters, what is the effective security of the resulting scheme? What parameters should one select to yield a desired, target effective security?

Our research contributes the following to the area of reasoning about the effective security of a security scheme:

- We formalize a method for reasoning about cryptographic proofs using attack trees;
- We expand attack tree analysis to include cryptographic reductions and provide algorithms for maintaining and automatically reasoning about these enhanced, cryptographic attack trees;
- We provide a proof-of-concept software tool, implemented in Python and leveraging the SymPy symbolic solver library; and
- We validate our tool using the Schnorr public-key signature scheme as a case study, comparing the results of our automated analysis with relevant, existing keylength recommendations.

1.1   Organization

In Chapter 2, we discuss existing methods for determining an appropriate keylength for a security scheme and review existing work on automated cryptographic proving. In Chapter 3, we formalize reasoning within attack tree structures, expand attack tree methodology to include cryptographic reductions and introduce our case study, the Schnorr signature scheme. In Chapter 4, we cover the concept of operations of our software tool and some characteristics we set to achieve with our design. In Chapter 5, we discuss our software tool implementation and discuss the algorithms driving the analysis engine of our software tool. We then examine some of the results of the software tool and compare these results with existing methods. In Chapter 6, we conclude and discuss future work.

CHAPTER 2:
Background

In this chapter, we review current keylength analysis techniques, the properties of cryptographic proofs required to perform a detailed keylength analysis and related research on automated proving for cryptographic constructions.

2.1 Keylength Analysis

Reasoning about cryptographic parameters requires an mathematical analysis relating the schemes' security parameters to the effective security of the resulting scheme. When a scheme is argued to have effective security of n bits given some choice of parameters, this characterizes the cost of attacking the scheme as equivalent in cost to brute-forcing some notional, symmetric system with an n-bit secret key, i.e., an effort of approximately 2^n operations [1]. For many schemes, there is a discrepancy between the security parameter and effective security for that scheme, based on the details of the construction and its proof of security. For example, 3DES uses three keys each of length 56 bits yet it is well-understood this construction does *not* yield a block cipher with 168-bits of effective security (3×56 bits); instead, 3DES has been shown to have an effective security of 112-bits. More efficient attacks against any system may exist, utilizing vulnerabilities in its implementation, in the properties of its underlying building-blocks, in its environment or via its users [1]. The scope of our analysis is to those properties and attacks covered by the security proof associated with the scheme.

Cryptography requires a security proof relating parameters to effective security. Both proof construction and proof analysis entails domain-expert knowledge and can be a laborious and nuanced process. New questions may not be able to re-use the results of prior analysis (i.e., when the prior assumptions were not general or when the context is slightly different). Instead, new questions may require whole-cloth re-analysis.

When organizations determine policy or select cryptographic parameters, they may find it too costly to conduct an in-depth proof analysis. Instead, they rely on guidance published by academic, government and private organizations such as the National Institute of Standards

and Technology (NIST), the Bundesamt fur Sicherheit in der Informationstechnik (BSI), the National Security Agency and the Internet Engineering Task Force. For example, NIST presents "best practice" parameters codified in look-up tables in publications which undergo periodic update, every two to four years [2]. During this period, the cost of some attacks may reduce, e.g., because of the increase in computer performance, decrease in resource cost or the discovery of new attack methods [1]. While the effects of Moore's law can be projected, the effect of new attacks on a system's effective security may not be apparent without a whole-cloth re-analysis. These impacts may not be disseminated until guidance is updated, if it is updated. In general, these published "best practice" parameters are intended to be relatively conservative; however, as periodic review is a human process, opportunities to re-think guidance is perhaps not as timely as circumstances may warrant. Beyond the possibility of being outdated, the limited context of the analysis codified in this guidance makes it inappropriate to leverage to answer new questions regarding security.

Referring to one of these "best practice" key length publications is certainly more accessible than a complete re-analysis, but it can be cumbersome as well. Furthermore, these guidelines are not always in agreement, creating a new hardship involving deciding upon which cryptographic guidance is most appropriate. Giry aggregates the keylength guidance from organizations, providing this as a resource through an interactive website [3]. It allows one to quickly compare the results of each report against one another to aide organizations in making a decision about security parameters [3]. While this reduces the workload associated with reading and interpreting this guidance, by putting each in relatively comparable terminology, the timeliness of information and resolving discrepancies remain as problems.

2.2 Methods for Achieving Provable Security

To date, it has been common to describe reductionist security proofs in one of two ways, as asymptotic guarantees or as concrete guarantees. The asymptotic setting was arguably first employed by Rabin in 1979 [4] and has been in relatively common use since. In a reductionist security argument, a proof of security involves a reduction between breaking the security of the system to solving a computationally hard problem [5]. In the asymptotic setting, this reduction yields statements such as "for a sufficiently large security parameter λ, no polynomial-time adversary has more than non-negligible probability of success in breaking the security property for the scheme" [5]. In the concrete setting, introduced

by Bellare and Rogaway, the goal is to produce more *practice-oriented* statements [6]. A concrete proof of security yields statements such as "an attacker, within time t and employing q queries to a problem-solving oracle O, is able to break the security property for the scheme under security parameter λ with at most ϵ probability of success" [7].

Each type of proof has advantages and disadvantages. For example, it appears some schemes are simpler to prove secure in the asymptotic setting. It is not the goal of our research, however, to criticize these two paradigms or to determine if one setting is more useful than the other, generally. The purpose of our software tool is to provide accurate answers regarding security parameters and effective security; thus, we have chosen to reason in the concrete security setting. This allows us to precisely characterize the cost of security due to the tightness of proof reductions and it allows us to interpret more accurate values for attack cost for given security parameters [8].

2.3 Automated Proof Generation

The academic community is continuously striving to create new cryptographic proof techniques, firmly grounded in mathematical foundations, to provide more easily verifiable guarantees with stronger results when analyzing cryptographic constructions [9]. Building on the work of Dolev and Yao [10], Kilian and Rogaway [11], Bellare and Rogaway [12] and others, the community has sought new methods to verify the correctness of machine-readable cryptographic proofs and to generate cryptographic proofs, in either a fully automatic or machine-assisted manner.

For some examples, Cortier and Warinschi [9] employ an existing tool called Casrul for automatically generating sound proofs in the computational model. Barthe et al. introduce EasyCrypt, a tool for machine-assisted security proofs from proof sketches using the CertiCrypt framework and available proof verifiers [13]. Blanchet introduces ProVerif, a cryptographic protocol verifier in the formal model [14]. Blanchet later extends this tool as CryptoVerif, an automatic protocol prover sound in the computational model [15] demonstrated to prove secrecy for a number of key exchange protocols.

Our software tool does not provide automatic proof generation, rather providing automatic analysis derived from proofs. The goal of our software tool is to employ concrete security proofs to determine guidance about security parameters or deriving effective security. Our

software tool uses metadata about concrete proofs to reason about security parameters and effective security. In the future, it may be possible to bridge automatic proof generation and our software tool. It may be possible to analyze the proofs from automated tools, or interpret it directly using a modified version of our software tool. This would allow the chain of automation to be extended from proof generation to including reasoning about effective security and recommending parameters.

CHAPTER 3:
Cryptographic Attack Tree Analysis Model

The target outcome of this research is a proof-of-concept software tool able to reason about effective security, automatically and dynamically, given different parameter options. It uses machine-readable proof and attack metadata to provide accurate answers regarding security parameters and effective security within a security scheme. Here, we propose a method to analyze this metadata based on attack trees. Since our methodology is based on attack trees, our software tool inherits many benefits traditionally associated with attack trees, e.g., extensibility and modularity.

We discuss the model and concepts traditionally associated with attack trees in Section 3.1. Our methodology expands typical attack tree analysis, however, to consider symbolic constraints related to proof tightness and attack costs. Inclusion of those constraints related to proof metadata, however, requires attack cost calculations associated with traditional attack trees to become more complicated, especially when multiple reductions may exist between attack objectives, which we discuss in Section 3.2. We present a case study for this methodology in Section 3.3, employing Schnorr's digital signature scheme.

3.1 Attack Trees

To better understand the weaknesses within a system, one can examine the system based on known methods of attacking it. An *attack tree* is a method of representing attacks against the system in a tree structure, allowing analysis of attack dependencies and costs. This method has been popularly discussed by Schneier and is considered mature [16]. Several projects exist allowing security researchers to employ attack trees for formalized red-team brainstorming and prioritizing system weaknesses, including ADTool [17], AttackTree [18], and SeaMonster [19].

In a typical attack tree, a goal G is related to a set of subgoals, $s_1, \ldots, s_j$, each represented as *nodes* in a directed graph (see Figure 3.1). To achieve goal G, an attacker must satisfy its subgoal dependencies. When two subgoals must both be satisfied to achieve the goal, its edges are labeled with a boolean "and" expression to express this dependency. Unless

otherwise noted, edges represent subgoals in an "or" relationship, where satisfying any subgoal achieves the goal. We denote an edge connecting goals G and dependency subgoal s_i as $G \rightarrow s_i$.
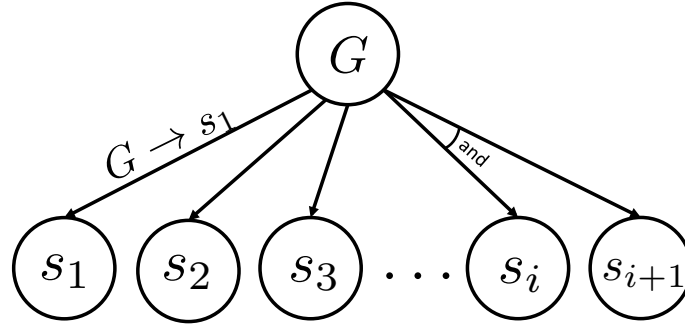
Figure 3.1: A simple attack tree, showing the relationship between goals and subgoals.

Figure 3.1 is a representation of a small attack tree. In a larger tree, each subgoal s_i may have subgoals itself, modeled as the children of node s_i, and so on. When the attack tree is complete, the objective G can be achieved by finding a set of paths from the leaves to the root, while satisfying the boolean restrictions attached to the edges involved in that path. In the remainder of our work, we do not utilize any "and" edges in our attack trees and all subgoals are in an "or" relationship. Thus, for this case, accomplishing G is equivalent to following a path from any leaf to G.

Consider the following example proposed by Schneier [16]. A thief wishes to access a 3-hour safe (G). The thief can pick the lock (s_1), use a blow torch to cut into the safe (s_2) or guess the combination (s_3). Achieving any of these subgoals is sufficient to achieve the goal G, but each have different costs, in terms of money, time, skill, etc. Attaining each subgoal may, itself, incur some prerequisites (e.g., buying the blowtorch), which can be modeled as further subgoals.

In order to characterize the relative security of the system, one can annotate the nodes of the tree to support attack cost-analysis. These annotations may represent the time or money required to achieve a subgoal. The values assigned to nodes are not limited in any fashion: one can assign a value indicating probability of attack success, level of attack complexity, pre-requisite skills or required equipment [16]. These values allow for flexible reasoning

8

about the security of the system. With these annotations, one can use the attack tree to reason about which attack may be launched with the highest probability of success and the lowest complexity, or the analysis of which attack is the cheapest [16]. We refer the reader to Schneier for further explanation of the utility of attack tree methodology for modeling computer systems [16].

When calculating the cost to execute a successful attack against the system in question, one must make some assumptions about the adversary potentially performing the attack. In this work, we assume the adversary is capable of utilizing any attack path and will choose the path with the minimum cost. Thus, no subgoal is impossible to achieve and we consider all node annotations to characterize *cost*, in some sense. We denote the cost to reach node x as $\text{Cost}(x)$. Given edge $x \rightarrow y$, the cost to achieve goal x using subgoal y is $\text{Cost}(x \rightarrow y)$. In many cases, $\text{Cost}(x \rightarrow y) = \text{Cost}(y)$. Consider, however, our prior example: if cost is dollars, y is "renting a blowtorch for an hour," x is "attacking a 3-hour safe" and $\text{Cost}(y) = \$100$, then it may be the case that $\text{Cost}(x \rightarrow y) = \$300$. To calculate the minimum cost to the attacker for achieving G, we consider the minimum cost attack terminating at G:

$$\text{Cost}(G) = \min_{j} \{\text{Cost}(G \rightarrow s_j)\}. \tag{3.1}$$

For example, in Figure 3.2 we show that the lowest cost associated with any of the attack paths from the subgoals of G is $\text{Cost}(G \rightarrow s_1)$. As a result, we have propagated that value, in this case 20, to $\text{Cost}(G)$. We can apply this procedure for calculating costs from the leaves of a completed tree all the way to the root to arrive at the overall cost to break the system.
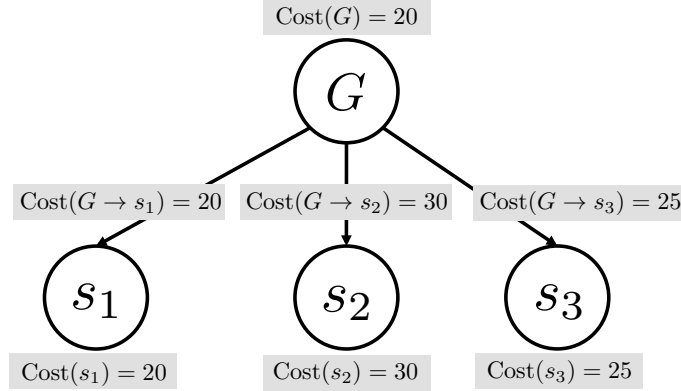
Figure 3.2: An attack tree with cost annotations.

3.2 Attack Trees with Reductions

We expand traditional attack trees, allowing a node to be annotated with an arbitrary set of *parameters* and an edge to be associated with a set of *reductions* (see Figure 3.3).
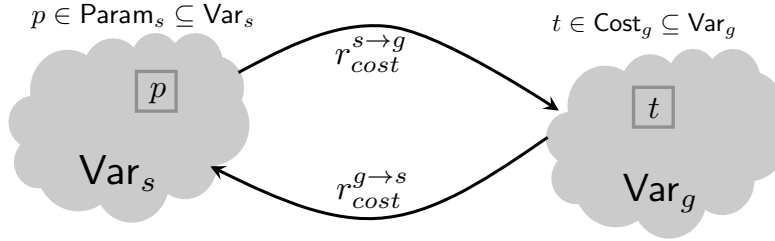
Figure 3.3: Reductions as relations between variables associated with sub-goals.

Let node s be associated with the set of free variables Var_s, which can be partitioned into two types: security parameters (Param_s) and cost parameters (Cost_s). For example, $p \in \mathsf{Param}_s \subseteq \mathsf{Var}_s$ may be a variable characterizing key length and $\tau \in \mathsf{Cost}_s \subseteq \mathsf{Var}_s$ may be a variable related to time complexity. Every variable $\sigma \in \mathsf{Var}_s$ has an associated domain $\mathsf{Dom}(\sigma)$ over which this free variable ranges and, given any finite set of values in this domain, the minimum and maximum are well-defined. Edge $g \to s$ may be annotated with a reduction r. A reduction r is a group of relations among Var_s and Var_g:

$$r = \left\{ r_{cost}^{s \to g}, r_{cost}^{g \to s}, r_{param}^{s \to g}, r_{param}^{g \to s} \right\} \qquad \text{where } r_{cost}^{g \to s} \text{ is a relation } f : \mathsf{Var}_s \to \mathsf{Cost}_g$$

For example, $r_{cost}^{g \to s}$ is a relation expressing cost, in terms of variables from Cost_g, to achieve g by employing subroutine s and instantiating its variables Var_s.

Reduction r can be interpreted as a guarantee that an adversary, given a subroutine solving s, will incur a cost greater than or equal to some guaranteed minimum cost (GMC) to solve g. We define the GMC indicated by reduction r when subroutine s is instantiated with variables p as:

$$\text{GMC}_r^{g \to s}(p) = r_{cost}^{g \to s}(p) \tag{3.2}$$

Returning to our prior example, consider an experiment to determine the minimum time to access a particular safe with a blow torch. The analysts are given some characteristics p for the blow torch (e.g., it uses acetylene gas, burns at 6000°F). The experiment predicts a conservative lower-bound for attack time to be at least two hours. The experiment is a type of reduction, specific to that goal (g), employing a general attack using a blowtorch (s) in the context of that torch's parameters (p) yielding two hours as the GMC.

As a result of ongoing efforts to improve or tighten security reductions, there may be several reductions relating subgoals (see Figure 3.4). Returning to our example, this is analogous to new experiments yielding improved bounds on the time to open the safe with the blowtorch, finding it takes at least three hours based on a more careful analysis, rather than at least two hours. We note that had experiments revealed that the safe could be opened in one hour using the blowtorch, this would not be a *tighter* reduction but, in fact, a *refutation* of the prior reduction, demonstrating it to have been unsound. In our model, we assume all our reductions are mathematically correct and sound.
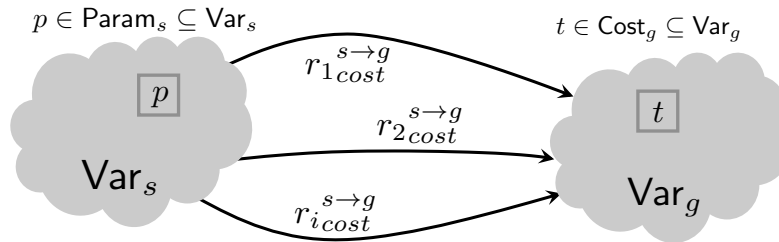
Figure 3.4: Each reduction yields a different relation, and may yield a different GMC.

3.2.1 Propagating Values "Up" the Tree

Let $R(g, s) = \{r_1, \ldots, r_i\}$ be the set of all reductions relating s to g. Each reduction r_i may yield a different GMC under conditions $p \in \mathsf{Var}_s$, yielding different relations for cost (see Figure 3.4). Given multiple reductions yielding different GMCs, we must determine which GMC to utilize in determining the cost along that particular attack path. Assuming all reductions are sound, the greatest lower bound (infimum) yields the overall GMC expressing cost for edge $g \to s$. The cost to achieve goal g via the attack path using subgoal s is therefore the maximum over all GMCs for reductions relating s to the cost of solving g:

$$\mathrm{GMC}(g \to s) = \max_i (\mathrm{GMC}_{r_i}^{g \to s}) \tag{3.3}$$

In order to calculate the cost of achieving G when dealing with multiple attack paths to G, one must first determine the GMCs of each reduction relating each subgoal to the goal (from Equation 3.2). Then, the cost along each attack path is resolved by taking the maximum along that path (from Equation 3.3). Finally, we determine the cost to achieve G (from Equation 3.1).

For example, consider the example tree from Figure 3.5 and its related information from Figure 3.6.

Figure 3.5: A simple attack tree, showing multiple reductions.

- $s.t \in \mathsf{Cost}_s \subseteq \mathsf{Var}_s$, a variable related to time complexity
- $u.t \in \mathsf{Cost}_u \subseteq \mathsf{Var}_u$, a variable related to time complexity
- $g.t \in \mathsf{Cost}_g \subseteq \mathsf{Var}_g$, a variable related to time complexity
- r_1 has relation $s.t \leq 2^{10} \times g.t^3$
- r_2 has relation $s.t \leq 2^{20} \times g.t^2$
- r_u has relation $u.t = 2 \times g.t$

Figure 3.6: Symbols and relations for the tree in Figure 3.5.
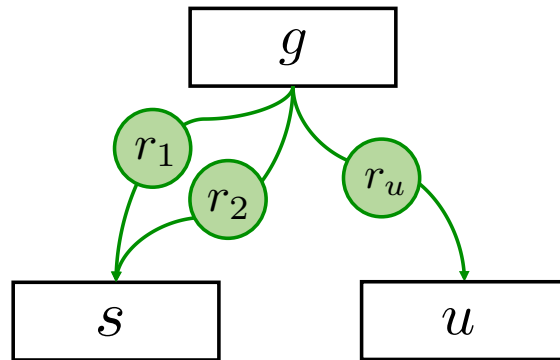
If we find that $s.t = 2^{80}$, r_1 yields $g.t \geq 2^{20}$ and r_2 yields $g.t \geq 2^{30}$. So, reduction r_1 states that goal g is secure against an adversary running in time $< 2^{20}$ while reduction r_2 states that goal g is secure against an adversary running in time $\leq 2^{30}$. We choose the value yielded by reduction r_2 (applying Equation 3.2) because it makes the stronger security claim. We then can say that the cost to achieve goal g via the attack path using subgoal s is 2^{30}. If we find $u.t = 2^{80}$, r_u yields $g.t = 2^{79}$. So, we apply Equation 3.1 to determine the cost to achieve goal $g = 2^{30}$.

3.2.2 Propagating Values "Down" the Tree

Thus far, we have discussed and given examples of propagating values "up" the attack tree, from subgoals to goals. We now discuss propagating values "down" the attack tree from goals to subgoals. There are differences that surface when propagating values "down" the tree. These differences are mainly in how security parameters and costs are reduced between two nodes and how we choose which values will get propagated into a node from its edges.

When traversing down the tree, instead of employing the guaranteed minimum cost, we employ the guaranteed peak cost (GPC). The GPC characterizes the smallest security parameters guaranteeing some lower-bound cost on attacker effort. Given $t \in \mathsf{Cost}_g$, we define GPC as:

$$\mathrm{GPC}_r^{g \to s}(t) = r_{param}^{g \to s}(t) = p \in \mathsf{Param}_s \tag{3.4}$$

When dealing with multiple reductions, each will produce a lower-bound on the time for attacker effort. Assuming all reductions are sound, the least upper bound (supremum) yields the smallest guaranteed attack cost for g given the ability to solve s. The cost to parameters to employ g via the attack path using subgoal s is therefore the minimum over all GPCs for

reductions relating g to the cost of solving s:

$$\text{GPC}(g \rightarrow s) = \min_i \left(\text{GPC}_{r_i}^{g \rightarrow s} \right) \tag{3.5}$$

Again, consider the example tree in Figure 3.5 and related information from Figure 3.6. If we are given $g.t = 2^{80}$, r_1 yields $s.t \leq 2^{250}$ and r_2 yields $s.t \leq 2^{180}$. So, reduction r_1 states that, given an adversary breaking g in time 2^{80} then there exists an adversary breaking s in time $\leq 2^{250}$, while reduction r_2 states that, under the same condition, there is an adversary breaking s in time $\leq 2^{180}$. We choose the value yielded by reduction r_2 because is portrays the tighter bound and represents the stronger adversarial ability. We can then resolve these two reductions and claim the cost to achieve subgoal $s = 2^{180}$, when given an adversary against g running in time 2^{80}. The reduction r_u yields $u.t = 2^{81}$. Since there is only one reduction between goal g and subgoal u, we can say the cost to achieve subgoal $u = 2^{81}$. Thus, to achieve an effective security of 2^{80} for g, we must select parameters forcing s to have effective security of at least 2^{180} and forcing u to have effective security 2^{81}.

3.3   Case Study: The Schnorr Signature Scheme

To express these ideas in the context of a non-trivial case study, we consider the Schnorr public-key signature scheme, with known relevant attacks and reductions. We selected this target scheme because it has the ability to demonstrate a variety of the features previously discussed for our methodology. In particular, the Schnorr signature scheme presents us the opportunity to reason about multiple reductions, given the original reduction presented by Schnorr [20] and the tighter reduction by Pointcheval and Stern [7]. It also allows us to demonstrate the situation where multiple attack subgoals must be analyzed, and subgoals are re-used within the tree in different contexts.

Employing the notation introduced thus far, we can express pictorially the attack tree for our Schnorr case study, annotated with variables and reductions (see Figure 3.7). The root, *Schnorr Signature*, represents a generic, high-level goal to break some aspect of the signature scheme. There are several notions of security applicable to the signature schemes— e.g., security against no-message attacks, security against chosen-message attacks—each yielding different trees. We have chosen to analyze one notion of security, existential forgery under an adaptively chosen-message attack (EF-ACMA).
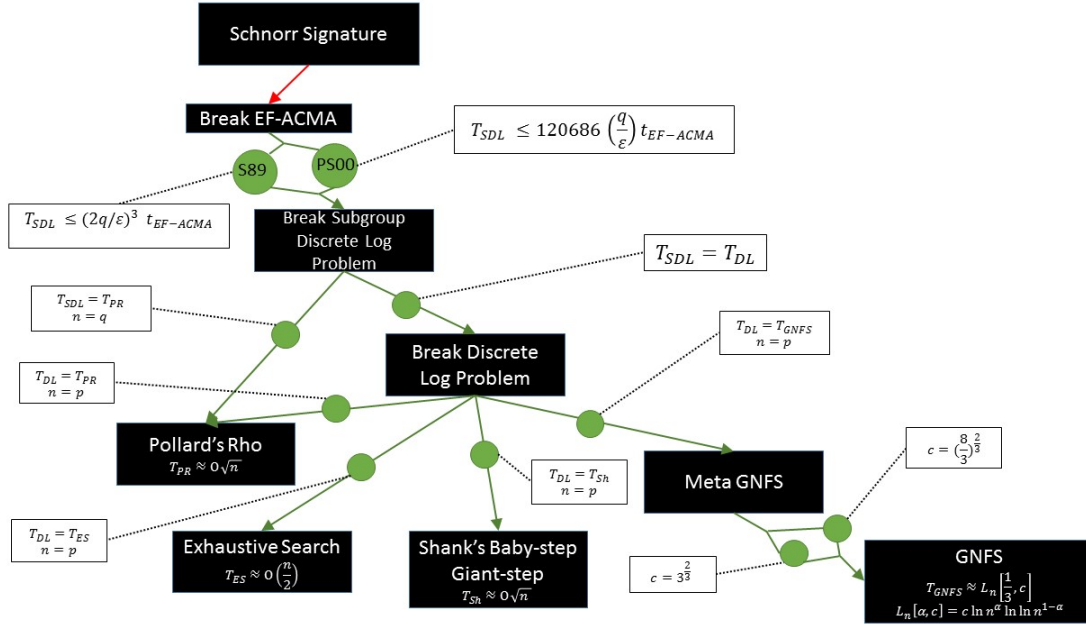
Figure 3.7: The Schnorr signature scheme attack tree. Nodes (black boxes) represent subgoals and edges (green lines) are annotated with reductions (green dots).

The subgoal of *Break EF-ACMA* is *Break Subgroup Discrete Log Problem*. The relationship between these goals are provided by two proofs: *S89*, the reduction originally provided Schnorr [20]; and *PS00*, the reduction provided by Pointcheval and Stern [7]. This is an example of the scenario discussed in Section 3.2. In this case, *PS00* provides the tighter bounds, improving a query bound from $O(q^3)$ to $O(q)$.

The *Break Subgroup Discrete Log Problem* has two subgoals: an attack via *Pollard's Rho* and reduction to a related problem, *Break Discrete Log Problem*. The *Break Discrete Log Problem* goal has subgoals, most of which are attacks or generalizations of attacks. One attack is, again, *Pollard's Rho*, demonstrating how a goal and its subgoals may share a common leaf.

The leaves in the tree each represent an attack, employing a specific algorithm to attack the properties of a goal. These are annotated with both variables and relations. Very general attacks like *Exhaustive Search* can be applied directly to most subgoals, but we omit those edges for brevity and clarity. The interested reader is referred to the following

15

resources for details on the attack nodes in this case study: *Pollard's Rho* algorithm [21], *Shank's Baby-step Giant-step* algorithm [22] and the general number field sieve (GNFS) algorithm [23].

CHAPTER 4:
Concept of Operations and Design

In this chapter we discuss concept of operations of our software tool to include propagating values in both directions, described briefly in Section 3.2. We discuss target features for our software tool and a high-level design supporting extensibility and modularity.

4.1   Concept of Operations

In its simplest use, our software tool should be capable of resolving two types of questions:

1. Given some set of security parameters (e.g., n, p and q), what effective security is provided?
2. For a desired target effective security, what security parameters should be used?

In the former case, the user provides n, p, and q and the tool returns the effective security λ. In the latter case, the user provides a target effective security λ and the program returns a set of satisfying parameters n, p and q.

Being able to re-run the above analyses enables interesting counter-factual analyses for prioritizing new research. This would enable us to pose questions, like, would improving the tightness bounds of this proof effectively strengthen the security of the scheme, or does some other factor create a bottleneck?

To illustrate the possible operation of the analysis engine consider our case study as an example (see Section 3.3). The user may desire to calculate the security parameters required to achieve a target level of effective security for the EF-ACMA property for the Schnorr signature scheme. The level of effective security can be expressed in terms of a lower-bound time required to break this property. Thus, the user provides some target value, like $t_{\text{EF-ACMA}} = 2^{80}$, as the time budget within which no adversary can break EF-ACMA. In response, the user expects the tool to provide recommended values of security parameters, such as p (the size of the prime-order group) and q (the size of the subgroup) for the scheme.

To accomplish this task, the analysis engine propagates the cost values down the tree as discussed in Section 3.2.2. The desired cost of $t_{\text{EF-ACMA}} = 2^{80}$ is passed through the two relations in the reductions leading to the subgoal *Break Subgroup Discrete Log Problem* (see Figure 3.7) to determine their GPCs as in the example provided in Section 3.2.2. The cost of *Break Subgroup Discrete Log Problem* is then determined utilizing the equations given in Chapter 3. In this case we derive $t_{\text{SDL}} = 2^{177}$ This procedure is then repeated to determine the costs for the subgoals of *Break Subgroup Discrete Log Problem*. For simplicity we opt not to step through each calculation down the tree. This process continues until we have propagated all cost values to each node from EF-ACMA to each of the leaves shown in Figure 3.7. We recall from Section 3.3 that each of these nodes is a generalized attack.

It is in these leaf nodes (attacks) where the cost value is directly translated into a security parameter that requires an attacker to undergo a particular cost. In other words, the leaf node reduces to no other problem. In our example, we have now populated the value $t_{\text{PR}} = 2^{177}$ into the Pollard's Rho attack. We can now utilize the approximation $t_{\text{PR}} = \sqrt{n_{\text{PR}}}$ to conclude $n_{\text{PR}} = 2^{354}$. This can be done for all leaf nodes.

Now that we have values for each security parameter in each leaf node, we can propagate those values up the tree as described in Section 3.2.1. Again, we opt not to step through every calculation. The user now has access to the value of every possible security parameter for every node in the attack tree that would make it impossible for an adversary to break the EF-ACMA property within the 2^{80} budget given. For instance, the user can now see that $p = 2^{354}$.

This example exhibits the need to make two passes through the tree (up and down) in order to provide values for parameters when given a target effective security. The process is also necessary when the analysis engine provides the effective security when given a set of security parameters.

4.2   Design Goals

Our software tool should reason about machine-readable proofs and attack metadata, to provide accurate conclusions regarding the relationship between security parameters and effective security. In addition, our software tool should be both *modular* and *extensible*.

Modularity allows expertise from one domain (e.g., using the general number field sieve to perform attacks against a cryptosystem) to be combined with that of another domain (e.g., the state of attacks against a particular hash function). It should be possible to develop modules independently, each defining their own symbols, cost parameters and relations. A new system employing both number-theoretic assumptions and hash functions may, then, employ these modules to reason about complex schemes. It should be possible to extend the tool, to incorporate new reductions, new objectives and new attacks. An extensible design allows organizations to incorporate new information for prompt, informed reactions to newly published attacks.

4.3 Software Design

We identify two base classes, related directly to our use of attack trees for reasoning about data: the *AttackEdge* and *AttackNode* classes. Figure 4.1 is a detail of the *AttackEdge* base class and its subclasses. Figure 4.2 is a detail of the *AttackNode* base class and its subclasses.

4.3.1 Attack Edges

The *AttackEdge* class represents the edge in a directed graph, and has an attribute named *parent* and *child* representing the nodes it connects. Each of these attributes reference an appropriate *AttackNode* class instance (see Section 4.3.2). *Reduction* is a subclass of *AttackEdge*, representing an edge annotated with a reduction. The *Reduction* subclass has a number of attributes: *full_name* is intended to be assigned a string that describes the reduction; *expr* lists the expressions describing how values are mapped between the goal and subgoal; *symbols_list* lists the symbols used in the reduction; *conservative_substitutions* lists a set of conservative substitutions for potential free variables in the reduction.
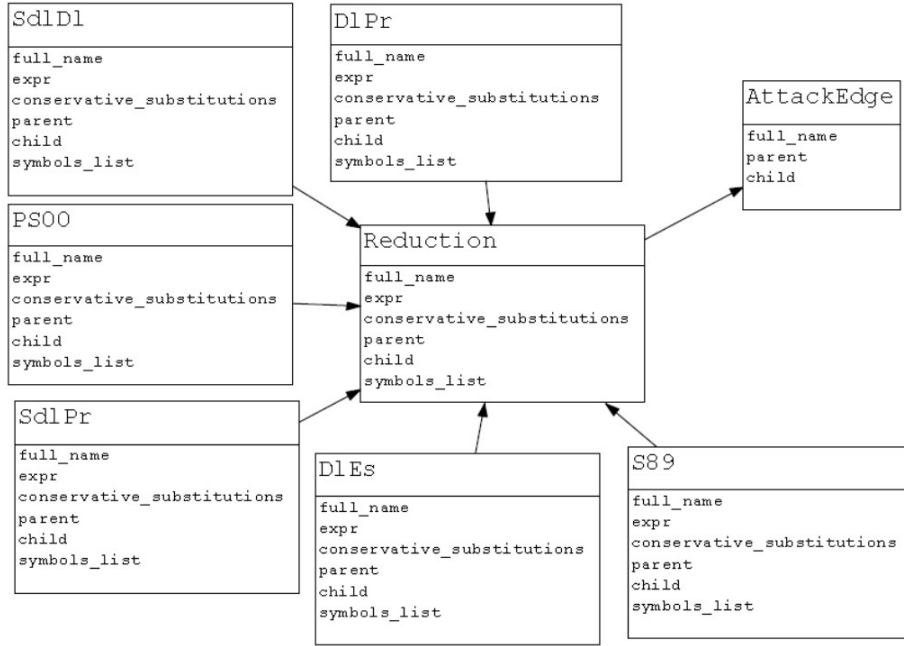
Figure 4.1: UML diagram for the *AttackEdge* class.

The remaining classes in Figure 4.1 are example subclasses of the *Reduction* class containing metadata specific to our case study. For example, *PS00* is the Pointcheval and Stern reduction between *Break EF-ACMA* and *Break Subgroup Discrete Log Problem*. *DlPr* is the reduction between *Break Discrete Log Problem* and *Pollard's Rho*.

4.3.2 Attack Nodes

The *AttackNode* base class has the *full_name* and *symbols_list* attributes, and they serve the same role as in the *AttackEdge* class. Our attack tree manages two types of nodes: *Objective* nodes and *Attack* nodes. All *Attack* nodes are leaf nodes, i.e., do not have children. All *Objective* nodes represent an objective or subgoal, such as "solving the discrete log problem." An *Objective* node may have one or more children (of any *AttackNode* type) and one or more parents (of *Objective* node type).

Figure 4.2: UML diagram for the *AttackNode* class.

The *Attack* class has the attribute *expr*, a list containing one or more expressions relating cost parameters and security parameters. An example of an expression contained within the *expr* list is $T_{\text{PR}} = \sqrt{n}$ from the *Pollard's Rho* node in Figure 3.7. In comparison, the *Objective* class has the attribute *constraints*, a list of zero or more expressions describing any constraints on the symbols within the objective. For example, in our Schnorr case study, the *Break Subgroup Discrete Log Problem* node employs the constraint $q < p$, expressing that the order of the subgroup q must be less than the order of the group p.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:

Implementation

In this chapter we discuss how our software tool and its routines are implemented. In Sections 5.1–5.4, we introduce notation and describe the routines for our analysis engine. In Section 5.5, we discuss the software implementation of this analysis engine. In Section 5.6, we provide sample analyses, compared with existing guidance.

5.1 Notation

The pseudocode expressing the algorithms for building and maintaining our data structures employs the following notation.

Graph Notation. Each graph $G = (N, E)$ is a directed, acyclic multigraph. We let $E(G)$ denote the edge multiset and $N(G)$ denote the node set.

Nodes. There are two distinct types of nodes, *Objective* nodes and *Attack* nodes (see Section 4.3.2). We let $O(G)$ denote the set of *Objective* nodes and $\mathcal{A}(G)$ denote the set of *Attack* nodes, where $N(G) = O(G) \bigcup \mathcal{A}(G)$ and $O(G) \bigcap \mathcal{A}(G) = \emptyset$

Edges. We let $\mathcal{R}(G)$ denote the set of reductions associated with G. Every edge is annotated with a reduction, so $|E(G)| = |\mathcal{R}(G)|$. For $e \in E(G)$, we let $\mathcal{R}(e) \in \mathcal{R}(G)$ be the reduction associated with edge e. Alternative notation for edge $e = (x, y)$ and its reduction $r = \mathcal{R}(e)$ include $x \to y$, $x \xrightarrow{r} y$ and $r(x, y)$.

Predecessor and Successor. Given edge $x \to y \in E(G)$, we say x is an *immediate predecessor* (parent) of y, and y is an *immediate successor* (child) of x. For edge $e = (x, y)$, we denote $x = e$.parent and $y = e$.child. Similarly, we denote $x \in y$.parents for the set of immediate predecessors and $y \in x$.children for the set of immediate successors. More generally, we say n_i is a *predecessor* of n_j if there exists a series of edges $n_i \to n_{i+1}$, $n_{i+1} \to n_{i+2}$, ..., $n_{j-1} \to n_j$ connecting n_i to n_j. By definition, if n_i is a predecessor of n_j, then n_j is a *successor* of n_i.

Symbols. For $x \in \mathcal{R}(G) \bigcup O(G) \bigcup \mathcal{A}(G)$, x has an attribute x.symbols. This is a list of symbols associated with the node (see Section 4.3.1 and Section 4.3.2). The variant attribute x.symbols$_{cost}$ and x.symbols$_{param}$ selects the symbols from x.symbols that are cost or security parameter symbols, respectively. For different nodes $n \neq m$, symbols

are unambiguous, i.e., m.symbols $\cap$ n.symbols $= \emptyset$. For any reduction $x \xrightarrow{r} y$, symbols are consistent with adjacent nodes, i.e., r.symbols $\subseteq x$.symbols $\cup y$.symbols.

Relations. For $x \in \mathcal{R}(G) \cup O(G) \cup \mathcal{A}(G)$, x has an attribute x.relations. This is a list of formulae expressed in terms of constants, relations and symbols in x.symbols. When $x \in O(G)$, x.relations is synonymous with the x.constraints attribute (see Section 4.3.2). When $x \in \mathcal{R}(G) \cup \mathcal{A}(G)$, x.relations is synonymous with x.expressions attribute (see Section 4.3.1 and Section 4.3.2).

Values. For $x \in O(G) \cup \mathcal{A}(G)$, x has an attribute x.values. This is a list of values associated with x.symbols. Initially, this list has no values, i.e., x.values $= \{(s, \perp)|$ for $s \in x$.symbols$\}$. When a value for symbol s is derived, then $(s, v) \in x$.values and s is no longer considered a *free symbol*.

Expressions. Each expression x has a number of attributes and support functions. The attribute x.free_symbols yields the set of free symbols in expression x. The function x.substitute(s, s') syntactically replaces the symbol s with symbol s'. The function x.solve(s) solves expression x for the value for free symbol s.

Assignment. In our pseudocode, $x \leftarrow y$ is the assignment operator and modifies the original object x. For lists, we abuse this notation to denote both insertion and update, depending on context. For example if $S \leftarrow \emptyset$, then $S \leftarrow (x, y)$ updates the list to be $S = \{(x, y)\}$. Subsequent insertions with related tuples, e.g., $S \leftarrow (x, y')$ and $S \leftarrow (x', y'')$, update the list to be $S = \{(x, y'), (x', y'')\}$.

5.2   Overview

Our software tool first builds the *Master Graph G* of reductions and the objectives defined in its database. Given a target objective σ_0, we prepare the data structure used for analysis, called the *Traceback Graph*, using the following steps:

1. Given target objective $\sigma_0 \in O(G)$, derive the subgraph of G comprised of σ_0 is its successors to form W, the *Working Graph* (see Figure 5.1).
2. Using W, initialize T, the *Traceback Graph* (see Figure 5.2).
3. Normalize T so that it has a tree structure, by duplicating those subtrees with multiple parents (see Figure 5.3).

```
 1: function SUBGRAPH(graph G, target σ₀ ∈ O(G))
 2:     graph W ← (∅, ∅)
 3:     for all n ∈ N(G) where σ₀ ∈ n.parents do
 4:         add n to N(W)
 5:         for all e ∈ E(G) where e = (n, n') do
 6:             add e to E(W)
 7:         end for
 8:     end for
 9:     return W
10: end function
```

Figure 5.1: Pseudocode for the *Subgraph* function.

```
 1: function INITTRACEBACK(graph W)
 2:     graph T ← W
 3:     for all x ∈ R(T) ∪ O(T) do
 4:         x.values ← ∅
 5:         for all s ∈ x.symbols do
 6:             x.values ← (s, ⊥)
 7:         end for
 8:     end for
 9:     return T
10: end function
```

Figure 5.2: Pseudocode for the *InitTraceback* function.

```
 1: function NORMALIZETRACEBACK(graph T, target σ₀ ∈ O(G))
 2:     for n ∈ N(T) where |n.parents| > 1 and depth(n, σ₀) is max over N(T) do
 3:         d ← |n.parents|
 4:         remove n from N(T)
 5:         create d copies of n and its successors to produce subtrees p₁, ..., p_d
 6:         add p₁, ..., p_d to T
 7:         modify all edges (x_i, n) ∈ E(T), so that the i-th edge is (x_i, p_i)
 8:         T ← NORMALIZE(T)
 9:     end for
10:     return tree T with root σ₀
11: end function
```

Figure 5.3: Pseudocode for the *NormalizeTraceback* function.

The *Traceback Graph* is used to hold values associated with the symbols for objectives and reductions in the *Working Graph*, derived as we traverse up or down the attack tree. For the

Master Graph and *Working Graph*, it is possible for a node to have multiple parents. Thus, before normalization, traversing down the *Traceback Graph* might cause nodes to be visited twice, via entirely different parents. Since derived values will be based on attributes of both parent and child, it would be incorrect to discard or update a value based on information from an unrelated parent. Thus, the *Traceback Graph* is normalized to form a tree, ensuring each node has only one parent. This is accomplished by duplicating subtrees that would otherwise be re-visited. After normalization, the *Traceback Graph* has the same number of edges as the *Working Graph* and is structurally identical to traversing the *Working Graph* in depth-first order. As an added benefit, the intermediate values generated as expressions and constraints are processed closely follows walking the *Working Graph*. Thus, the *Traceback Graph* allows one to "trace back" the symbols resolved and values derived at each step of walking the graph.

At this point, the attack tree is prepared for analysis. Analysis solves for either effective security (in terms of cost) or settings (in terms of security parameters) needed to achieve a particular level of security (see Section 4.1). For example, if a user wants to solve for the cost to break the objective, the user will provide the security parameters for the scheme as input. Since we must propagate values in both directions within our attack tree (see Section 4.1), analysis is a two-pass process. *PhaseOne* derives values from the root to the leaves, and *PhaseTwo* from the leaves to the root.

1. The user selects target symbols related to objective σ_0, placing these in the global *SOLVE* list.
2. The user builds K, a set of known values for other parameters related to σ_0.
3. We build *traverseList*, the list of $r \in \mathcal{R}(T)$, in depth-first order starting at σ_0.
4. *PhaseOne*: we utilize *Reductions* in T to populate values down the tree, where we will use the expressions in the *Attack* nodes to derive the values needed for analysis and save these in T (see Figure 5.4).
5. *PhaseTwo*: we utilize values stored in T during the first pass to solve for remaining free variables (see Figure 5.11).
6. If successful, a value $(s, v) \in \sigma_0$.values will exist for any symbol $s \in SOLVE$.

In the following sections, we elaborate on *PhaseOne* and *PhaseTwo* and the routines supporting these.

26

5.3 First Pass

During *PhaseOne*, we populate-down values from the root the leaves (see Figure 5.4). We begin by moving known values from K into root σ_0.

```
 1: function PHASEONE(tree T with root σ₀, known values K, traverseList)
 2:     for (s, v) ∈ K where s ∈ σ₀.symbols and (s, ⊥) ∈ σ₀.values do
 3:         σ₀.values ← (s, v)
 4:     end for
 5:     for r(σᵢ, σⱼ) ∈ traverseList do
 6:         r ← POPULATEKNOWNS(σᵢ, r)
 7:         r ← REMOVEFREE(r.values, r)
 8:         if σⱼ ∈ 𝒜(T) then
 9:             σⱼ ← REMOVEFREE(r.values, σⱼ)
10:         else if σⱼ ∈ O(T) then
11:             if there are multiple edges for (σᵢ, σⱼ) then
12:                 σⱼ ← RESOLVEMULTIEDGE(r.values, σⱼ, σᵢ)
13:             else
14:                 σⱼ ← RESOLVEOBJECTIVE(r.values, σⱼ)
15:             end if
16:         end if
17:     end for
18:     return T
19: end function
```

Figure 5.4: Pseudocode for *PhaseOne*.

We visit each edge in depth-first order, populating known values, using expressions to derive more known values, simplifying expressions and populating values to adjacent objectives. At each edge, values are populated from parent to edge (see Figure 5.5), changing these according to their symbol type.

Next, known values at edges are substituted into expressions, following the same process employed at attack nodes, i.e., leaves (see Figure 5.6). If there are enough known values to eliminate all but one free symbol, we will have effectively determined a value for that symbol. We can solve for the free variable and store the value.

```
 1: function POPULATEKNOWNS(node σ, r ∈ R(T))
 2:     for (s, v) ∈ σ.values and s ∈ σ.symbols ∩ r.symbols do
 3:         if PhaseOne then
 4:             r.values ← (s, v) ∈ σ.values
 5:         else if (s, v′) ∈ σ.values and s ∈ σ.symbols_cost and v < v′ then
 6:             r.values ← (s, v) ∈ σ.values
 7:         else if (s, v′) ∈ σ.values and s ∈ σ.symbols_param and v > v′ then
 8:             r.values ← (s, v) ∈ σ.values
 9:         end if
10:     end for
11:     return r
12: end function
```

Figure 5.5: Pseudocode for the *PopulateKnowns* function.

```
 1: function REMOVEFREE(vals, p ∈ R(T) ∪ A(T))
 2:     for expr in p.relations do
 3:         for (s, v) ∈ vals where s ∈ expr.free_symbols do
 4:             expr ← expr.substitute(s, v)
 5:         end for
 6:         if s ∈ expr.free_symbols and |expr.free_symbols| == 1 then
 7:             value ← ADVANTAGINGSUBSTITUTION(expr, s)
 8:             if PhaseOne and p ∈ R(T) and s ∉ SOLVE then
 9:                 p.values ← (s, value)
10:             else if (PhaseOne and p ∈ A(T)) or PhaseTwo then
11:                 p.values ← (s, value)
12:             end if
13:         else if p ∈ R(T) then
14:             p ← CONSERVATIVESUBSTITUTION(expr.free_symbols, p)
15:             if a new value was just uncovered by the previous step then
16:                 p ← REMOVEFREE(T, vals, p)
17:             end if
18:         else if P ∈ A(T) then
19:             error              ▷ analysis fails if attack nodes have too many free variables
20:         end if
21:     end for
22:     return p
23: end function
```

Figure 5.6: Pseudocode for the *RemoveFree* function.

The process for removing a free variable involves deriving values for symbols based on relationships in reductions (see Figure 5.7) and employing conservative symbol substitutions (see Figure 5.8).

```
 1: function ADVANTAGINGSUBSTITUTION(relation, symbol)
 2:     if relation is "=" then
 3:         exp ← relation
 4:     else if relation is either "<" or ">" then
 5:         exp ← GreaterSide(exp) = (LesserSide(exp) + 1)
 6:     else if relation is either "≤" or "≥" then
 7:         exp ← GreaterSide(exp) = LesserSide(exp)
 8:     end if
 9:     value ← exp.solve(symbol)
10:     return value
11: end function
```

Figure 5.7: Pseudocode for the *AdvantagingSubstitution* function.

```
 1: function CONSERVATIVESUBSTITUTION(free_symbols, r ∈ R(T))
 2:     if PhaseOne then                                              ▷ during PhaseOne
 3:         σ ← r.child
 4:     else                                                         ▷ during PhaseTwo
 5:         σ ← r.parent
 6:     end if
 7:     if σ.symbols ∩ free_symbols ≠ ∅ then
 8:         for expr ∈ r.conservative_substitutions do
 9:             for (s, v) ∈ r.values where s ∈ expr.free_symbols do
10:                 expr ← expr.substitute(s, v)
11:             end for
12:             if s ∈ expr.free_symbols and |expr.free_symbols| == 1 then
13:                 value ← ADVANTAGINGSUBSTITUTION(expr, s)
14:                 r.values ← (s, value)
15:             end if
16:         end for
17:     end if
18:     return r
19: end function
```

Figure 5.8: Pseudocode for the *ConservativeSubstitution* function.

Conservative substitutions are "safe" substitutions based on bounds, selecting parameters

that most advantage the adversary. This is often required when reductions employ under-constrained expressions or when free symbols have a bound but no known parameter (discussed in Section 4.3.1). Similar rationale is employed when deriving new values based on relations, re-arranging equations and selecting bounds that most advantage the adversary.

Next, after edge expressions are simplified and new values have been derived, we populate down values to child nodes (see Figure 5.9). The values stored at the objective node are updated, depending on the type of symbol. If the symbol represents a cost variable, the edge value updates the value if it is smaller, i.e., the adversary cost is smaller using the new edge. The opposite is true for security parameters, i.e., the new edge implies a larger parameter is required.

```
 1: function ResolveObjective(vals, σ ∈ O(T))
 2:     for (s, v) ∈ vals where s ∈ σ.symbols do
 3:         if (s, ⊥) ∈ σ.values then
 4:             σ.values ← (s, v)
 5:         else if (s, v′) ∈ σ.values and s ∈ σ.symbols_cost and v < v′ then
 6:             σ.values ← (s, v)
 7:         else if (s, v′) ∈ σ.values and s ∈ σ.symbols_param and v > v′ then
 8:             σ.values ← (s, v)
 9:         end if
10:     end for
11:     return σ
12: end function
```

Figure 5.9: Pseudocode for the *ResolveObjective* function.

For the case when multiple edges exist, i.e., due to multiple reductions, populating values down to the child node requires modification, as potential values populated at the node must be withheld until all reductions are processed (see Figure 5.10). The new values from each edge are held temporarily, this multi-edge value group is managed analogously to how values are updated at the child, before they are populated to the child. When the final edge is processed, the values are resolved as normal.

```
 1: function RESOLVEMULTIEDGE(vals, σ₁, σ₂)
 2:     if this is the first (σ₁, σ₂) edge processed then
 3:         σ₁.mvals[σ₂] ← ∅                    ▷ holds values for edges between σ₁ and σ₂
 4:         σ₁.count[σ₂] ← total number of edges between (σ₁, σ₂)
 5:     end if
 6:     let multv ≡ σ₁.mvals[σ₂]                ▷ an alias, for notational convenience
 7:     let count ≡ σ₁.count[σ₂]                ▷ an alias, for notational convenience
 8:         ▷ now, process one edge, i.e., associated with the reduction annotated with vals
 9:     count ← count − 1
10:     for (s, v) ∈ vals where s ∈ σ₁.symbols do
11:         if PhaseOne then                                              ▷ during PhaseOne
12:             if (s, ⊥) ∈ multv then
13:                 multv ← (s, v)
14:             else if (s, v′) ∈ multv and s ∈ σ₁.symbols_cost and v < v′ then
15:                 multv ← (s, v)
16:             else if (s, v′) ∈ multv and s ∈ σ₁.symbols_param and v > v′ then
17:                 multv ← (s, v)
18:             end if
19:         else                                                          ▷ during PhaseTwo
20:             if (s, ⊥) ∈ multv then
21:                 multv ← (s, v)
22:             else if (s, v′) ∈ multv and s ∈ σ₁.symbols_cost and v > v′ then
23:                 multv ← (s, v)
24:             else if (s, v′) ∈ multv and s ∈ σ₁.symbols_param and v < v′ then
25:                 multv ← (s, v)
26:             end if
27:         end if
28:     end for
29:     if count == 0 then                                    ▷ this was, in fact, the last edge
30:         σ₁ ← RESOLVEOBJECTIVE(multv, σ₁)
31:     end if
32:     return σ₁
33: end function
```

Figure 5.10: Pseudocode for the *ResolveMultiEdge* function.

5.4 Second Pass

After our first, top-to-bottom pass where values are populated across the tree, a bottom-to-top pass is performed (see Figure 5.11). This *PhaseTwo* pass is similar to *PhaseOne* and utilizes the same support routines. To traverse back up the attack tree, we re-visit edges in

reverse order.

```
 1: function PHASETWO(tree T, traverseList)
 2:     for r(σ_i, σ_j) ∈ reverse(traverseList) do
 3:         if σ_j ∈ O(T) then
 4:             r ← POPULATEKNOWNS(σ_j, r)
 5:         end if
 6:         r ← REMOVEFREE(r.values, r)
 7:         if there are multiple edges for (σ_i, σ_j) then
 8:             σ_i ← RESOLVEMULTIEDGE(r.values, σ_i, σ_j)
 9:         else
10:             σ_i ← RESOLVEOBJECTIVE(r.values, n_i)
11:         end if
12:         for expr ∈ r.relations do
13:             for (s, v) ∈ r.values where s ∈ expr.free_symbols do
14:                 expr ← expr.substitute(s, v)
15:             end for
16:         end for
17:     end for
18:     return T
19: end function
```

Figure 5.11: Pseudocode for *PhaseTwo*.

Attack nodes should now be fully populated with values, and those values can be populated up the tree from child to parent. Values for objectives and multi-edges are resolved analogously to prior logic. As described in Section 3.2, walking up the tree in the presence of multiple reductions requires slightly different logic: assuming all reductions are sound, when the parent's cost is determined to be greater, we consider this to be a tighter proof and select the more accurate result (with analogous reasoning when the more accurate reduction allows us to select a smaller security parameter).

5.5   Python Implementation

We implement our software tool in the Python programming language version 3.5.1 [24] using the Anaconda package and environment manager [25]. We select Python because it is open source, widely used and relatively accessible, all of which support the extensibility goals of our software tool. To implement our attack tree data structures, we employ the

NetworkX Python module version 1.11 [26]. Of interest to us, NetworkX supports directed, multi-graphs and provides support functions to iterate through graph components, to assign and manage data associated with nodes and edges, to traverse graphs and to export graphs to formats usable by graph visualization software. To manipulate expressions associated with objectives, attacks and reductions, we employ the Sympy Python module [27]. This provides support for expression re-writing and symbolic solving.

5.6   Case Study: The Schnorr Signature Scheme

We validate our software tool using the Schnorr case study introduced in Section 3.3. Figure 5.12 is a graphical representation of the tree—its objectives, reductions and attacks—comparable to Figure 3.7. The software producing this visualization does not draw explicit multi-edges, and represents edge direction using line thickness rather than arrow heads.
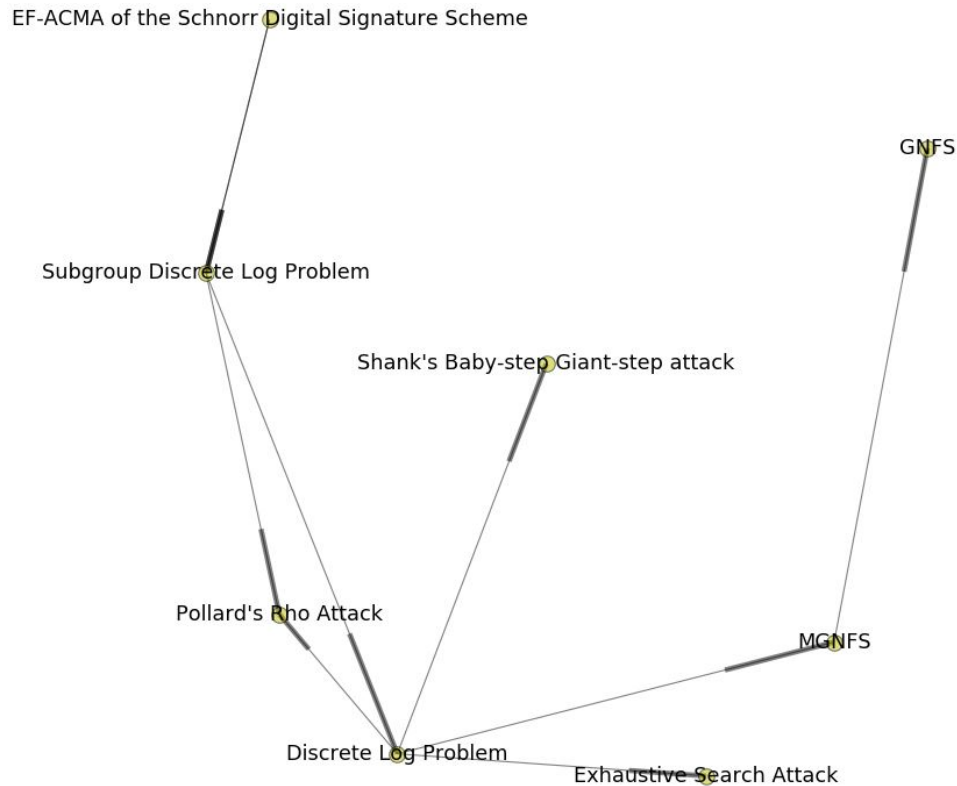
Figure 5.12: Graph for the Schnorr case study, generated by our software tool.

We consider two analysis cases in the context of this case study, both employing the target objective *Break EF-ACMA* for σ_0, the root of our analysis tree.

Case 1. We supply our software tool with a set of security parameters and use it to solve for the cost to break the Schnorr scheme in the EF-ACMA sense. This requires setting (i) $SOLVE$ to include the relevant time parameter for σ_0 and (ii) known values that include relevant security parameters for σ_0. We are essentially asking the question "what is the effective security for EF-ACMA with the Schnorr signature scheme using the chosen security parameters?"

Case 2. We supply our software tool with a set of cost parameters and use it to solve for security parameters for the Schnorr scheme. This requires setting (i) $SOLVE$ with relevant security parameters and (ii) known values that include 2^ℓ as the target time-cost parameter. We are essentially asking "under what parameters will the Schnorr signature scheme achieve an effective security of 2^ℓ for EF-ACMA?"

The full list of symbols associated with attacks and objectives in the subgraph associated with components connected to σ_0 is given in Figure 5.13.

```
Symbols for node Exhaustive Search Attack

        Es.n:       Keylength
        Es.t:       Time


Symbols for node Shank's Baby-step Giant-step attack

        BsGs.n:     Keylength
        BsGs.t:     Time


Symbols for node Discrete Log Problem

        Dl.p:       Order of the Group in Discrete Log
        Dl.t:       Time to break Discrete Log


Symbols for node GNFS

        Gnfs.c:     constant
        Gnfs.t:     time
        Gnfs.n:     n


Symbols for node EF-ACMA of the Schnorr Digital Signature Scheme

        EfACma.R:   Number of queries to the signing oracle
        EfACma.e:   Probablility of success of the attack
        EfACma.T:   Time to break Schnorr Signature under EF-ACMA
        EfACma.q:   Number of queries to the random oracle


Symbols for node MGNFS

        MGnfs.n:    n
        MGnfs.t:    time


Symbols for node Pollard's Rho Attack

        Pr.t:       Time
        Pr.n:       Keylength


Symbols for node Subgroup Discrete Log Problem

        Sdl.t:      Time to break Subgroup Discrete Log
        Sdl.p:      Order of group in Subgroup Discrete Log
        Sdl.q:      Order of subgroup in Subgroup Discrete Log
```

Figure 5.13: Attack and objective symbols for the Schnorr case study.

For Case 1, a sample run of our software tool is provided in Figures 5.14 and 5.15, showing values associated with each node and edge. In this case, the level of effective security is

determined to be $\approx 2^{31.6}$ when using the following security parameters: the order of the group is 2^{1024}, the order of the subgroup is 2^{160}, the probability of success to 1.0, the number of queries to the both *Break EF-ACMA* random oracles as $2^{31.5}$, the number of queries to the random oracle within the Schnorr reduction to 2^{19}. In reality, the random oracle parameters are inferable based on running time bounds; however, we defer to future work enhancements to employ conservative symbolic substitution—i.e., symbolic re-writing of expressions based on conservative relations—rather than our more simple approach of conservative value substitution.

```
The values for node EF-ACMA of the Schnorr Digital Signature Scheme are as follows:

        EfACma.R    2^31.5000000000000
        EfACma.e    1
        EfACma.T    2^31.6191011974573
        EfACma.q    2^31.5000000000000


The values for node Subgroup Discrete Log Problem are as follows:

        Sdl.t       2^80
        Sdl.p       2^1024
        Sdl.q       2^160


The values for node Discrete Log Problem are as follows:

        Dl.p        2^1024
        Dl.t        2^86.7661192502810


The values for node MGNFS are as follows:

        MGnfs.n     2^1024
        MGnfs.t     2^86.7661192502810
```

Figure 5.14: Case 1 output, showing node data when solving for cost.

```
The values for edge EF-ACMA <=> Subgroup Discrete Log (Pointcheval and Stern) are as follows:

        EfACma.R   2^31.5000000000000
        EfACma.T   2^31.6191011974573
        Sdl.q      2^160
        Sdl.t      2^80
        EfACma.e   1
        EfACma.q   2^31.5000000000000
The values for edge EF-ACMA <=> Subgroup Discrete Log (Schnorr) are as follows:

        Sdl.t      2^80
        EfACma.e   1
        EfACma.T   2^20
        EfACma.q   2^19
The values for edge Subgroup Discrete Log <=> Discrete Log are as follows:

        Sdl.t      2^86.7661192502810
        Sdl.p      2^1024
        Dl.p       2^1024
        Dl.t       2^86.7661192502810
The values for edge Discrete Log <=> Exhaustive Search are as follows:

        Es.n       2^1024
        Es.t       2^1023
        Dl.p       2^1024
        Dl.t       2^1023
The values for edge Discrete Log <=> Pollard's Rho are as follows:

        Pr.t       2^512
        Dl.p       2^1024
        Pr.n       2^1024
        Dl.t       2^512
The values for edge Dl <=> Meta GNFS are as follows:

        Dl.p       2^1024
        MGnfs.n    2^1024
        MGnfs.t    2^86.7661192502810
        Dl.t       2^86.7661192502810
The values for edge M <=> G are as follows:

        Gnfs.c     1.92299942707654
        Gnfs.t     2^86.7661192502810
        MGnfs.n    2^1024
        MGnfs.t    2^86.7661192502810
        Gnfs.n     2^1024
The values for edge Discrete Log <=> Shank's Baby-step Giant-step are as follows:

        BsGs.n     2^1024
        Dl.p       2^1024
        BsGs.t     2^512
        Dl.t       2^512
The values for edge Subgroup Discrete Log <=> Pollard's Rho are as follows:

        Sdl.t      2^80
        Pr.t       2^80
        Sdl.q      2^160
        Pr.n       2^160
```

Figure 5.15: Case 1 output, showing edge data when solving for cost.

For Case 2, a sample run of our software tool is provided in Figures 5.16 and 5.17, showing values associated with each node and edge. Providing $2^{31.5}$ as the time bound for node *Break EF-ACMA* corresponds to declaring that the best-known attacks breaking EF-ACMA must take a time exceeding $2^{31.5}$, which provides 31.5 bits of effective security. The analysis yields that, to ensure this level of security, one must choose the order of the subgroup to be at least $\approx 2^{160}$.

```
The values for node EF-ACMA of the Schnorr Digital Signature Scheme are as follows:

            EfACma.R    2^31.5000000000000
            EfACma.e    1.00000000000000
            EfACma.q    2^31.5000000000000
            EfACma.T    2^31.5000000000000


The values for node Subgroup Discrete Log Problem are as follows:

            Sdl.q       2^159.761797605085
            Sdl.p       2^159.761797605085
            Sdl.t       2^79.8808988025427


The values for node Discrete Log Problem are as follows:

            Dl.p        2^159.761797605085
            Dl.t        2^79.8808988025427
```

Figure 5.16: Case 2 output, showing node data when solving for security parameters.

```
The values for edge EF-ACMA <=> Subgroup Discrete Log (Pointcheval and Stern) are as follows:

        EfACma.R    2^31.5000000000000
        EfACma.e    1.00000000000000
        EfACma.q    2^31.5000000000000
        EfACma.T    2^31.5000000000000
        Sdl.q       2^159.761797605085
        Sdl.t       2^79.8808988025427


The values for edge EF-ACMA <=> Subgroup Discrete Log (Schnorr) are as follows:

        EfACma.e    1.00000000000000
        Sdl.t       2^79.8808988025427
        EfACma.q    2^31.5000000000000
        EfACma.T    2^31.5000000000000


The values for edge Subgroup Discrete Log <=> Discrete Log are as follows:

        Sdl.p       2^159.761797605085
        Dl.p        2^159.761797605085
        Sdl.t       2^79.8808988025427
        Dl.t        2^79.8808988025427


The values for edge Discrete Log <=> Exhaustive Search are as follows:

        Es.n        2^80.8808988025427
        Dl.p        2^80.8808988025427
        Es.t        2^79.8808988025427
        Dl.t        2^79.8808988025427


The values for edge Discrete Log <=> Pollard's Rho are as follows:

        Pr.n        2^159.761797605085
        Dl.p        2^159.761797605085
        Dl.t        2^79.8808988025427
        Pr.t        2^79.8808988025427


The values for edge Discrete Log <=> Shank's Baby-step Giant-step are as follows:

        BsGs.t      2^79.8808988025427
        Dl.p        2^159.761797605085
        Dl.t        2^79.8808988025427
        BsGs.n      2^159.761797605085


The values for edge Subgroup Discrete Log <=> Pollard's Rho are as follows:

        Pr.n        2^159.761797605085
        Sdl.q       2^159.761797605085
        Sdl.t       2^79.8808988025427
        Pr.t        2^79.8808988025427
```

Figure 5.17: Case 2 output, showing edge data when solving for security parameters.

While testing our software tool, we discovered it was capable of solving for time within the GNFS attack, however solving for parameters proved more complex. We suspect the problem involved limits associated with the symbolic solver: SymPy appears to lack solvers able to handle computing n, given T and c, in the super-polynomial sub-exponential relationship for attacks employing the general number field sieve:

$$T = \exp(c \ln n^{1/3} \ln \ln n^{2/3}).$$

As a result, the node for the GNFS attack and the edge connecting it to the rest of the tree are missing from Figures 5.16 and 5.17. We leave for future work further verification of this problem and investigation into solutions or alternatives for this scenario.

Comparing the results of our analyses to existing guidance, Figure 5.18 displays a selection of the key-length guidance for schemes based on the security of the discrete log problem, equating these to the cost of security for a symmetric scheme (i.e., characterizing its effective security). The Schnorr signature scheme employs discrete log assumptions. The "key" and "group" parameters from Figure 5.18 can be equated to the symbols *Sdl.q* and *Sdl.p*, respectively, in our case study (see Figure 5.13).

Method	Date	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash
Lenstra / Verheul	2027	91	2299 1856	161	2299	172	181
Lenstra Updated	2018	80	1329 1478	160	1329	160	160
ECRYPT II	2011 - 2015	80	1248	160	1248	160	160
NIST	2010 (Legacy)	80	1024	160	1024	160	224
ANSSI	2014 - 2020	100	2048	200	2048	200	200
NSA	-	-	-	-	-	-	-
RFC3766	-	80	1233	160	1233	148	-
BSI	2015	128	2048	224	2048	224	224

Figure 5.18: Parameters for discrete log schemes, for key size 160 and a discrete log group size 1024. Source [3]: D. Giry. (2015). BlueKrypt-cryptographic key length recommendation. [Online]. Available: http://www.keylength.com.

Given a 160-bit subgroup and 1024-bit group, our software tool returns an effective security of 80 bits for the *Break Subgroup Discrete Log Problem* security objective (see Figure 5.14), matching the effective security for many recommendations from Figure 5.18. Given 224-bit subgroup and 2048-but group, our software tool returns an effective security of 112 bits; interestingly, this is less than that claimed by the BSI recommendation from Figure 5.18. Using a 128-bit target effective security, Figure 5.19 summarizes the discrete log parameters from existing guidance and Figure 5.20 shows our results.

Method	Date	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash
Lenstra / Verheul	2076	129	6790 5888	230	6790	245	257
Lenstra Updated	2090	128	4440 6974	256	4440	256	256
ECRYPT II	2031 - 2040	128	3248	256	3248	256	256
NIST	> 2030	128	3072	256	3072	256	256
ANSSI	2021 - 2030	128	2048	200	2048	256	256
NSA	-	256	-	-	-	384	384
RFC3766	-	128	3253	256	3253	242	-
BSI	2015	128	2048	224	2048	224	224

Figure 5.19: Parameters for discrete log schemes, comparable in strength to 128-bit symmetric encryption. Source [3]: D. Giry. (2015). BlueKrypt-cryptographic key length recommendation. [Online]. Available: http://www. keylength.com.

```
The values for node Subgroup Discrete Log Problem are as follows:

        Sdl.t       2^128
        Sdl.q       2^256
        Sdl.p       2^256


The values for node Discrete Log Problem are as follows:

        Dl.t        2^128
        Dl.p        2^256
```

Figure 5.20: Results from our software tool using 128-bit effective security.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusion

We have described the current complexity and methodology to reason about cryptographic schemes, using reductionist proofs and data about known attacks for the practice-oriented goal of parameter selection. State-of-the-art is to employ generic, conservative recommendations published periodically, which are not necessarily current or informed by recently disclosed attacks. We developed a proof-of-concept software tool capable of reasoning about keylength for cryptographic schemes using machine-readable proof metadata. In support of this, we formalized reasoning about cryptographic adversaries within the attack tree model, and expanded attack trees to include proof reductions.

For this tool, we described a concept of operations, use cases and design goals. We argued the software tool should be able to provide recommendations about keylength when given a set of parameters, as well as provide parameters necessary to achieve a target effective security. From there we presented the software design to achieve the stated goals and provided the algorithms to implement the software. We validate our software tool utilizing the Schnorr public-key signature scheme as a non-trivial case study.

While our prototype had some limitations unrelated to our methodology, overall we showed that automating reasoning about keylength and effective security is achievable. When the attack and proof metadata is provided in a machine-readable format, our software tool achieved the desired goals of providing accurate and timely information specific to a particular scheme. Additionally, modularity and extensibility were achieved by designing the software tool in a way that allows experts to contribute knowledge from their domain, connecting it to a graph of other proof knowledge and facilitating its use in an automated framework.

6.1 Future Work

Further investigation into the limitations of our implementation, as presented in Section 5.6, is warranted. It will be necessary to verify that the limitations stem from the solver's inability to handle particular situations, rather than a mistake in the implementation of

our software tool. Then, once the problem is properly framed, the implementation can be refitted. We also believe it is fruitful to expand the library of attacks and objectives to include other case studies employing concrete security reductions. Lastly, work can be done to bridge automatic proof generation as explored in prior work, and automatic proof analysis as explored here.

List of References

[1] A. K. Lenstra, "Key lengths," Wiley, Tech. Rep., 2006.

[2] E. Barker and Q. Dang, "Recommendation for key management part 3: Application-specific key management guidance," NIST, Gaithersburg, MD, Tech. Rep. Special Publication 800-57 Pt 3 Rev 1, Jan. 2015.

[3] D. Giry. (2015). BlueKrypt-cryptographic key length recommendation. [Online]. Available: http://www.keylength.com

[4] M. Rabin, "Digitalized signatures and public-key functions as intractable as factorization," Massachusetts Institute of Technology, Cambridge, Mass, Tech. Rep. MIT/LCS/TR-212, Jan. 1979.

[5] S. Micali and L. Reyzin, "Improving the exact security of digital signature schemes," *Journal of Cryptology*, vol. 15, no. 1, pp. 1–18, 2002.

[6] M. Bellare and P. Rogaway, "The exact security of digital signatures-how to sign with rsa and rabin," in *Advances in Crytology-EUROCRYPT '96*, U. Maurer, Ed. Berlin, Germany: Springer-Verlag, 1996, pp. 399–416.

[7] D. Pointcheval and J. Stern, "Security arguments for digital signatures and blind signatures," *Journal of cryptology*, vol. 13, no. 3, pp. 361–396, 2000.

[8] B. Blanchet and D. Pointcheval, "Automated security proofs with sequences of games," in *Advances in Cryptology-CRYPTO 2006*. Springer-Verlag, 2006, pp. 537–554.

[9] V. Cortier and B. Warinschi, "Computationally sound, automated proofs for security protocols," in *Programming Languages and Systems*, M. Sagiv, Ed. Berlin, Germany: Springer-Verlag, 2005, pp. 157–171.

[10] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[11] J. Kilian and P. Rogaway, "How to protect des against exhaustive key search," in *Advances in Cryptology—CRYPTO'96*, N. Koblitz, Ed. Berlin, Germany: Springer-Verlag, 1996, pp. 252–267.

[12] M. Bellare and P. Rogaway, "The game-playing technique," *International Association for Cryptographic Research (IACR) ePrint Archive: Report*, vol. 331, pp. 1–29, 2004.

[13] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology–CRYPTO 2011*, P. Rogaway, Ed. New York, USA: Springer, 2011, pp. 71–90.

[14] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *CSFW-14*. Washington, D.C.: IEEE Computer Society Press, 2001, pp. 82–96.

[15] B. Blanchet, "A computationally sound mechanized prover for security protocols," *Dependable and Secure Computing, IEEE Transactions on*, vol. 5, no. 4, pp. 193–207, 2008.

[16] B. Schneier, "Attack trees," *Dr. Dobb's Journal*, vol. 24, no. 12, pp. 21–29, 1999.

[17] B. Kordy, P. Kordy, S. Mauw, and P. Schweitzer, "Adtool: security analysis with attack–defense trees," in *Quantitative Evaluation of Systems*.

[18] isograph, Alpine, UT. (2015). *AttackTree*. [Online]. Available: http://www.isograph. com/download-products/. Accessed June 12, 2016.

[19] SINTEF,Trondheim, Norway. (2013). *SeaMonster*. [Online]. Available: https://sourceforge.net/projects/seamonster/. Accessed June 12, 2016.

[20] C.-P. Schnorr, "Efficient identification and signatures for smart cards," in *Advances in cryptology—CRYPTO'89 proceedings*. Berlin, Germany: Springer-Verlag, 1989, pp. 239–252.

[21] J. M. Pollard, "Monte carlo methods for index computation $(mod\backslash p)$," *Mathematics of computation*, vol. 32, no. 143, pp. 918–924, 1978.

[22] H. Cohen, *A course in computational algebraic number theory*. New York, NY: Springer Science & Business Media, 2013, vol. 138.

[23] D. M. Gordon, "Discrete logarithms in gf(p) using the number field sieve," *SIAM Journal on Discrete Mathematics*, vol. 6, no. 1, pp. 124–138, 1993.

[24] Python Software Foundation, Columbia, DE. (2015). *Python 3.5.1*. [Online]. Available: https://www.python.org/downloads/release/python-351/. Accessed Apr. 15, 2016.

[25] Continuum Analytics, Inc., Austin, TX. (2015). *Anaconda 4.0.0*. [Online]. Available: https://www.continuum.io/. Accessed Apr. 15, 2016.

[26] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, Aug. 2008, pp. 11–15.

[27] Sympy Development Team, Los Alamos, NM. (2016). *Sympy 1.0: Python library for symbolic mathematics*. [Online]. Available: http://www.sympy.org. Accessed Apr. 15, 2016.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California